Porting High-Performance Applications to Itanium 2

Tommy Hoffner and Patrik Sandahl

ERICSSON

TAKING YOU FORWARD

Overview

- The Application
- The Porting Experience
- Unexplored Features
- Toolchain
- Knowledge Gaps
- Conclusion

Ericsson

- Ericsson is one of the world's leading providers for telecommunications solutions.
- Solutions for mobile and wireline telephony.
- Supports IP based as well as circuit switched traffic.
- Ericsson embraces a variety of microprocessors, operating systems and other sourced building blocks for its platforms and solutions.

The Application

- The execution engine in an emulated proprietary hardware platform on modern hardware.
- The current strategy is Just-In-Time binary translation.
- The platform is a component in Ericsson's mobile network solutions.
- It's primary mobile telephony usage is extremely demanding on:
  - Processing power.
  - Large physical memory space.
  - Backward compatibility.

Maintaining the Legacy

- Programs for the platform are written in a proprietary programming language
  - The existing code base consists of many million lines of code.
- They are compiled to binary machine code for the original proprietary microprocessor
  - CISC, special purpose and unsigned integer only instruction set.
  - No explicit memory addresses in the machine code, all accesses pass through a hardware symbol table.
- The platform supports patching a running system without interruption. Patches are written in the proprietary machine code language.
- ’Padded cell’ execution environment for application code
  - Fine grained fault detection and recovery.

Heritage and Opportunities

- The previous generation of the system was based on the Alpha microprocessor.
- Complex control flow.
- We have tried standard batch compilation.

- As we only target a few programs we can use domain knowledge to the extreme.
- We target one specific SKU of Itanium 2.

The Execution Model

- Code is executed in ’jobs’ – short snippets of code.
- The job scheduler pick jobs from a queue that usually are a few entries deep.
- Extremely cheap ’context switching’ between jobs.
- Code can make both synchronous and asynchronous calls
  - An asynchronous call will result in the insertion of an entry in the job queue.

The Binary Translation Model

- Compilation units (similar to extended basic blocks) in the source application are translated when executed the first time.
- If the compilation unit ends with a conditional branch the translator temporarily pauses the translation at that point and executes the generated code in order to get a runtime descision for the branch. After the descision is made the translation resumes.
  - Conditional branches are transformed to favor the common case.
  - Not yet taken branches have ’dangling exits’ that activates the compiler when taken. I.e. non executed compilation units in the source application do not result in code generation.
- As translation is interleaved with job execution the performance of the translator has effects on the job’s response time.

The Binary Translation Model

- The generated code is incrementally tied together in strings as the translator walks through the path of execution.
- A path of execution in the source program usually generates a highly specialized code string
  - Several paths of execution can join and follow the same string if they share common code.
- Code can be invalidated and retranslated, e.g. due to inserted patches or activated tracing in the source program or due to profiling for dynamic optimizations.
  - Invalidations are also made periodically to clean the code cache and to adapt to new behaviour

Approaching the Itanium 2

- When the migration to Itanium 2 was started the architecture was perceived as a ’nightmare’ to compile for, e.g.:
  - Itanium 2’s in-order execution model and the need for explicit bundling required the introduction of a new - and expensive – instruction scheduling pass to the translator.
  - Highly complex instruction set.
  - The lack of an indexed addressing mode.
  - Every rule had an exception and every exception was described by a rule.
- But …

Good Experiences

- Dual mode instruction scheduling:
  - High throughput mode focusing on performance for the translator.
  - High quality mode focusing on performance in the generated code.
- Extensive support on architectural and implementational details.

The Perfect Match #1

- The dynamic translation model together with good knowledge about application behavior allows the translator to make really good choices for branch instruction completers:
  - .dptk, .dpnt, .sptk and .spnt.
  - .many and .few.
- Getting the .many/.few selection right was one of the biggest performance improvers.

The Perfect Match #2

- Memory instructions with locality hints for control of data in the cache hierarchy is a perfect match for the extreme level of application knowledge used in the platform.
- The strong PMU implementation - with the D-Ears as high-lights – allows the translator’s embedded dynamic profiling engine to propose, insert and evaluate data prefetches.

Other Nice Features Used

- Post increment/decrement. Had tremendous impact on the code footprint when we got it right.
- Long moves simplified the materialization of 64-bit addresses.
- Predicated execution. Can make code really straight and dense.

Not so Perfect Match

- Huge code footprint combined with the job scheduler’s knowledge about which job to execute next was thought to be the perfect match for instruction prefetch but wasn’t …
  - Instruction prefetch seems to be restricted to code that resides somewhere in the cache hierarchy and to addresses that are in the TLB.
  - Cumbersome interface; Why not a lfetch.i[.fault]?
- ’Missing’ templates
  - Inserted nops adds to code footprint.

Unexplored Features

- Register Stack
- Rotating Registers
- Floating Point
- Control Speculative Loads

Toolchain

- Ecc significantly better than the tested alternatives.
- Could do better on some basics though.
- We had some problems with idb initially.
- VTune doesn't support dynamically generated code and the supported methodology doesn't suite us.

Knowledge Gaps

- There are quite a lot of information on what to do if you want to get the most out of Itanium, but they are often costly to implement.
- Since the step to Itanium seems quite large for applications that needs to generate their own binary code, descriptions of simple and cheap ways to handle things like bundling and scheduling would have simplified our work. It would probably also have made us more comfortable in our initial work on the platform.
- Cheap scheduling and its performance relative to ’best in class’ would have been really useful.

Conclusion

- As compiler designers, we consider Itanium to be an architecture that will keep us busy for a long time.
- But we have changed our view on why
  – Before we started: ”A lot of work just to get things working”.
  – Now: ”a lot of interesting and useful features to tweak”.

ERICSSON

TAKING YOU FORWARD